Comput Syst Sci & Eng (1996) 5: 259–273

Computer Systems
Science & Engineering

Youssef H, Sadiq M. Sait, AlMulhelm AS, et al.•High-level synthesis from purely behavioral descriptions •
COMPUTER SYSTEMS SCIENCE AND ENGINEERING 11 (5): 259-273 SEP 1996

High-level synthesis from purely behavioral descriptions

Habib Youssef, Sadiq M Sait, A S Al-Mulhelm and M S T Benten

Department of Computer Engineering, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
Email: sadiq@ccse.kfupm.edu.sa

This paper presents an integrated system which accepts as input a purely behavioral description expressed in a subset of the C language and transforms it into RTL description in the AHPL language. The novelty of this work is the introduction of a new stack intermediate form, and the use of hardware-specific optimization during the scheduling phase. This idea has produced sizable decrease in the number of control steps compared to other reported techniques.

Keywords: high-level synthesis, compilers, RTL, optimization, scheduling, allocation, layouts, VLSI, simulation

1. INTRODUCTION

High-level synthesis is the process of translating an input behavioral description of a digital system into a structural description. Behavioral specifications of digital systems can be described using high-level programming languages, Hardware Description Languages (HDL), algorithmic pseudo codes, etc.[1, 2]. Several high-level synthesis systems are reported which use a wide variety of input specification languages ranging from PASCAL-like to ISPS[3, 4].

High-level synthesis is a complex process and consists of various transformations. Due to this complexity, the process of high-level synthesis is divided into tasks[5]. In each task, a partial refinement towards the target structure is introduced. The behavioral description of digital systems is compiled into an intermediate form[6]. This intermediate form abstracts the input behavioral description specification. Two generally used types of intermediate forms are parse trees and graphs[5]. The most common intermediate forms are graph-based models[6, 7]. Graph representations are attractive intermediate forms because many compiler optimization techniques can be applied on them[7]. Data flow graphs (DFG) and control flow graphs (CFG) are the most widely used types of graph-based intermediate forms.

This paper describes a high-level synthesis system that accepts an input description in a high-level language and generates the appropriate structural description. The behavioral language selected is a subset of the C programming language. The structural description is in A Hardware Programming Language (AHPL)[8, 9]. This work has two distinctive features:

1. A stack data structure is used as the primary intermediate form. To our knowledge this is the first time that techniques and data structures used by programming language interpreters are employed in the high-level synthesis of hardware.
2. The input behavioral specification is interpreted as a hardware description. Thus, besides the usual software optimization techniques applied during the synthesis tasks, hardware specific optimization is also performed.

Figure 1 gives the main steps followed by the high-level synthesis system. First, the behavioral description of the digital system modeled in C-like language is compiled into a stack representation in Blocked Reverse Polish Notation (BRPN). Second, a canonical register transfer language (CRTL) description is generated from this stack representation. The structure is then extracted from the graph-based model.

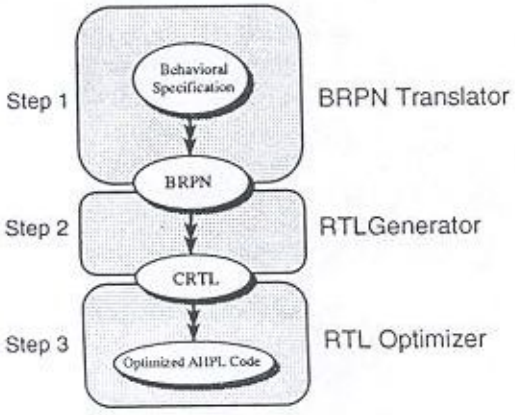


Figure 1 Major tasks in the new high-level synthesis system

Finally, the extracted structure is optimized. In the remainder of this paper, we shall provide a detailed description of each of these steps.

2. BLOCKED REVERSE POLISH NOTATION (BRPN)

The first step in high-level synthesis consists of the interpretation of the input behavioral description. The major concern in this step is to get a suitable internal representation. Optimization is not the issue. In this case, stacks are most appropriate.

The stack data structure is commonly used by interpreters. It allows easy and efficient evaluation of arithmetic expressions. In stack machines expressions are simplified into groups of basic operations. The basic operations within each group are executed sequentially. The advantages of using a stack representation as an intermediate form are:

1. simple, yet powerful data structure,
2. easy to generate and manipulate, and
3. flexibility of grouping stack operations.

The translation of C-like descriptions to stack notation is a language to language translation. The mapping of C-like to stack notation is illustrated with an example. Consider the following high-level language description of the *While* construct:

```
i = 1;
while (i ≤ k) do
      ⋮
      body
      ⋮
      i = i + 1
end-while.
```

The first line in the description is an assignment. This assignment is translated into pushing the constant '1' onto the stack followed by a pop operation. Then the popped value is loaded into register *i* (*push i*). The corresponding stack representation is '*push* 1; *pop i*'. In order to execute the loop, the condition '$i \leq k$' is evaluated. The corresponding stack representation is as follows: the first operand is pushed onto the stack '*push i*', then the second operand is pushed '*push k*'. Now both operands are on top of the stack. These operands are popped and the condition expression (k !< i) is formed. The evaluation of this condition will determine whether to execute the body of the loop or not. The stack representation corresponding to this is '*push i*; *push k*; !< 0', where '0' is the address of the body. The mapping of the rest of the code into stack representation produces the following stack code:

```
push 1; pop i                    assign 1 to register i
push i; push k; !< 0             evaluate the condition (i ≤ k)
begin (code segment_0:)
      ⋮
push i; push 1; + pop i          increment i by 1
push i; push k; ! < 0            if (i ≤ k) go to segment_0
end (code segment_0).
```

We define a block to be a sequence of stack operations with the restriction that transfer of control from other blocks can only take place to the first statement of the block. *Blocks* are delimited with square brackets and labeled. Each block label is prefixed with the letter 's'. Thus the above stack representation is written as follows:

```
push 1; pop i
push i; push k; !< 0
[push i; push 1; + pop i; push i; push k; ! < 0]s0.
```

Because of this blocking in the stack representation, the Reverse Polish Notation (RPN) code generated will be called *blocked RPN* (BRPN) to differentiate it from the traditional way of expressing stack operations as a sequence of statements.

In BRPN, the relation between the blocks is a *father-children* relationship. That is, during the execution of a set of stack instructions in a block, control may transfer to another block (calling block is termed as *father* and the called block as *child*). After a successful execution of the *child* block, control returns to the *father*. For the above example, the body of the *while* loop can be compiled in the same block *s0*. Another way is to write the body in another block *s1* where *s1 is called from s0 every time* the loop condition is evaluated true. This *father-child* relationship is illustrated in Figure 2. Recall that as a rule, the blocking strategy used here

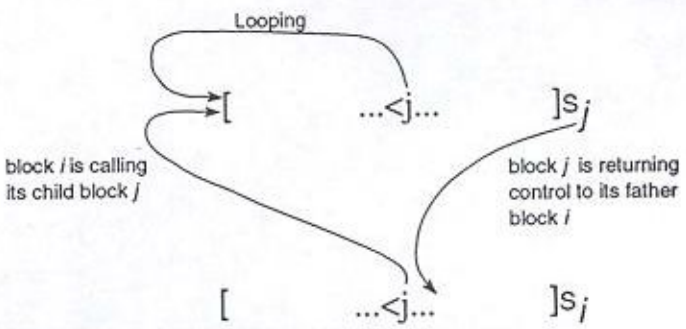


Figure 2 Father-children relation in the stack representation

should satisfy the condition that: *control can only transfer to the beginning of a block.*

The representation in BRPN is different from graph-based representations such as DFG and CFG. A DFG models the data dependencies among the operators and variables. Furthermore, a DFG may be disconnected. The precedence relations on these data operations cannot be shown in a DFG alone. These relations are modeled by a CFG. The DFG and CFG are linked to resemble the behavioral model. For example, Flamel[2] uses *Dacons*, which are control-data flow graphs (CDFGs). Each node represents a basic block. A basic block is the largest number of statements in the code that does not contain any control construct and has one entry point and one exit point. The edges in *dacons* represent the control transfer among the basic blocks.

On the other hand, the structure in BRPN resembles the structure of high-level language models. The basic entity in BRPN is also a block. However, unlike Flamel blocks, BRPN blocks may contain control statements and loops. The definition of a block here is the largest number of stack operations that satisfy the following condition: '*control can only be transferred to the beginning of a block*'. For this, blocks cannot be merged because this might violate the transfer of control condition. The precedence relation among the blocks is more like the strategy used in high-level language subroutine calling. This is illustrated in Figure 3. Optimization cannot be carried out on BRPN code. For example, a common sub-expression in different blocks cannot be eliminated, since each sub-expression is located in a different block and invoked at different instances. BRPN is used just as a primary intermediate form to capture all data and control flow information in the input behavioral specification. In a following step, the BRPN representation is transformed into a canonical register transfer language (CRTL) description on which various optimization tasks are performed.

3. GENERATION OF CRTL MODELS FROM BRPN

Graph-based intermediate forms are most suitable for optimization tasks. Therefore, before attempting to perform any optimization, we first transform the BRPN into an intermediate RTL represented as CDFG. This transformation is accomplished in a systematic way and with the same ease as the interpretation of the initial behavioral specification into BRPN.

Recall from Section 2 that the general block format in BRPN is:

[stack operations]sx

In the above format, x is a distinct label given to the block of stack operations. These stack operations may either be operations on data or control statements. Figure 4 shows some of the stack operations where *Oprnd* stands for operand, *Dopr* stands for data operation, *Copr* stands for control operation, and *Dltr* stands for a delimiter. More details are given later.

The stack operations in the above format are executed in sequence, that is, the leftmost operation in the block is the first and the rightmost is the last. The location of each stack operation, whether data or control, inside the code is unique. As a result, the above format can be divided into a sequence of statements where each statement holds either a data or control operation and has a unique tag. Moreover, all branches to successor statements are defined.

The above description has similarities to some known RTL formats such as AHPL except that each step is restricted to a simple data transfer or control transfer statement. Thus, the generated code is referred to as CRTL. This bears some similarity to Boolean minimization, where the Boolean function is first expressed as canonical sum of minterms prior to being minimized to produce a minimum sum of product terms.

There are two steps involved in generating the CRTL statements

1. Determining the CRTL statement number; this number is formed as a pair (BN,SN), where BN is

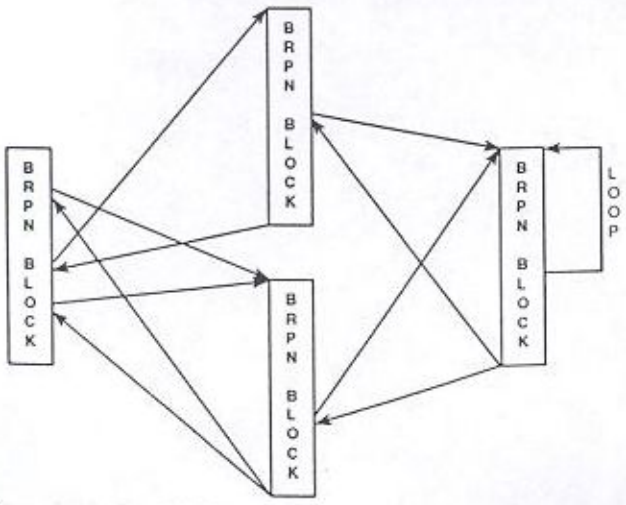


Figure 3 Blocking in BRPN

Operands	Opnd	lx	push x onto stack.
	Opnd	sx	pop top of stack and save in x
	Opnd	Constant	always pushed onto stack.
Data Operations	Dopr	d	duplicates top value of stack.
	Dopr	p	pops top value of stack and prints.
	Dopr	f	pops entire stack and prints.
	Dopr	q	exits program.
	Dopr	c	pops entire stack.
	Dopr	z	stack level pushed onto stack.
	Dopr	+	add
	Dopr	-	subtract
	Dopr	*	multiply
	Dopr	/	divide
	Dopr	%	remainder
	Dopr	^	power
			performs the operation on top two elements of stack and pushes the result back.
	Dopr	;	stack's top element is index to array.
	Dopr	:	used for array operation.
Control Operations	Copr	<x	top two elements of stack are popped and compared. Transfer is made to block x.
	Copr	>x	
	Copr	=x	
	Copr	!>x	
	Copr	!<x	
Delimiters	Dltr	[	marks the start of block.
	Dltr	]	marks the end of block.

Figure 4 Some stack operations and operands used in BRPN

the block number uniquely defined for each block of the stack code, and SN is the sequence number associated with any operation within the block that causes popping from the top of the stack; SN is the relative position of the operation from the left of the block.

2. Indicating whether the stack operation is a control or a data statement; if the stack operation is a data operation then the RTL statement is to hold the operation and the address to the next statement. Otherwise, the statement is to hold the condition of the operation and the addresses of the true and the false branches.

The format of a CRTL statement consists of:

stmnt no.	data operation	control operation	true branch	false branch

Using this format, each statement in the code obtained from BRPN holds either a data operation and an unconditional branch to the next statement in sequence or a conditional branch to a particular statement. The two steps defined to generate CRTL statements are graphically illustrated in Figure 5.

3.1 BRPN-to-CRTL algorithm

The algorithm that translates BRPN format to CRTL format is presented in Figures 6 and 7.

The main task of the algorithm is to extract the data and control parts. The input to the algorithm is the BRPN code translated from behavioral description. The algorithm outputs a generalized CDFG where nodes represent CRTL statements and edges represent the order of execution of these statements. The algorithm begins by reading a *token*. A token may be a delimiter (*Dltr*), an operand (*Opnd*), a data operation (*Dopr*), or a control operation (*Copr*). Delimiters

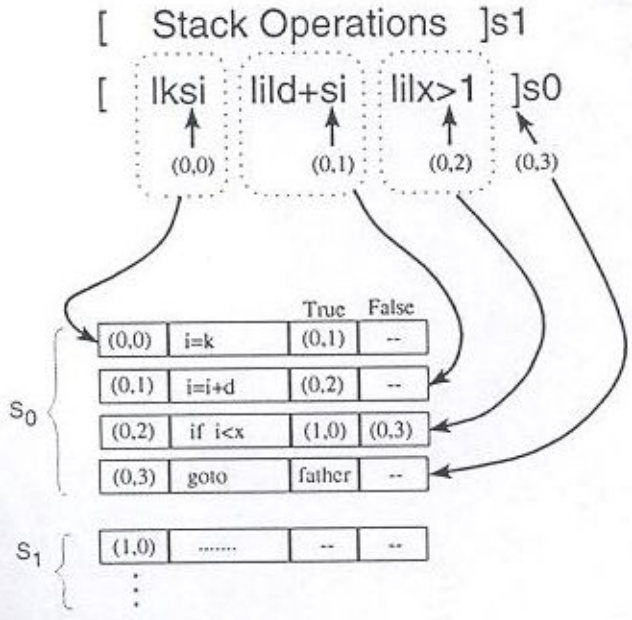


Figure 5 The translation of BRPN into CRTL

```
Current_block = -1
input(token)
WHILE (not eof)
  CASE token :
    token = '['          \* The starting of a block *\
      Initialize a new block with seq_num=0;
      Current_block=block label.
    token = push operation
      Push operand to the stack.
    token = pop operation
      Pop operand from the stack;
      Form the expression:  operand=expression
       and put it in a new link;
       set the address of this link to:
              (current_block, seq_num)
      Increment seq_num.
    token = data operation   \* construct expression *\
      data_operation(token).
    token = control operation
      control_operation(token).
      Increment seq_num.
    token = ']'          \* The ending of a block *\
      Increment seq_num;
      Current_block=-1.
  END_CASE
  input(token)
END_WHILE
```

Figure 6 The algorithm that generates CRTL code from BRPN

```
Procedure data_operation(token)
Begin_Procedure
    Pop the operands ;
    Construct the parse tree to retrieve the original
    expression;
    Push the retrieved expression back to the stack.
End_Procedure


Procedure control_operation(token)
Begin_Procedure
    Pop the operands;
    Construct the parse tree to retrieve the original
    expression;
    Assign the resulting expression to the control
    operation field;
    Set the address of this link to:
            (current_block, seq_num);
    Set the address of true branch to:
            (blk_num, 0);
    Set the address of false branch to:
            (current_block, seq_num+1);
    Determine father.        \* see text for methods of
End_Procedure                   father determination *\
```

Figure 7 Procedures used by the algorithm in Figure 6

are used to signal special events in the BRPN code, e.g. '[' and ']' signal the beginning and end of the block, respectively. Operands may either be numeric constants or variables. Constant operands can be of any length and are always pushed onto the stack. Each variable operand starts with either 'l' for pushing the variable onto stack or 's' for assign-

ing to the variable at the top of the stack. Data operator characters are one character tokens. On the other hand a control operator consists of one or two characters. Figure 4 shows some of the operators and operands used by BRPN. Details on the translation algorithm are given below.

3.2 Determination of branching conditions

The general format of a conditional operation in BRPN notation is as follows:

$$[\ldots \mathrm{I}Opnd_1 \, \mathrm{I}Opnd_2 \ \textbf{Rop BN} \ldots]s_i$$

In this statement, I is the push operation, $Opnd_1$ and $Opnd_2$ are operands to be pushed on top of the stack, **Rop** is a relational operation on the two top values of the stack, and **BN** is the block number to branch to on true. Now, if the condition holds (true) then a branch takes place to the first statement of block BN, that is statement (BN,0). While on false, a branch takes place to the next operation in sequence in the current block (current block number, current sequence number +1) as shown in Figure 5.

Example

In this example, we illustrate the generation phases of a structural description from a behavioral description. The first phase is to translate the behavioral model of a digital system into BRPN. To illustrate the process of translating algorithmic (behavioral) descriptions into BRPN, we use the bubble sort algorithm shown in Figure 8a.

Using the stack operations given in Figure 4, the BRPN code corresponding to the above algorithm is given in Figure 8b.

The second phase is to apply the algorithm presented in Figure 6 to generate CRTL from this BRPN. To illustrate the steps of the algorithm, consider the following two BRPN blocks from Figure 8b:

```
[lj;a st lj 1+;a lj:a ltlj 1+:a]s2
[lj;a lj 1+;a <2 ljd1+sj ljli>1]s1.
```

In the first line of the code '[' indicates the start of the block. The block number (blk_num) is derived from the block label. In this case, blk_num equals 2 and the sequence number (seq_num) is reset to zero. The next input token is 'lj;', causing the variable 'j' to be pushed into the stack. However, since 'j' is followed by ';', the variable is interpreted as an array index with the name of the array following the ';'. Thus, 'a[j]' is pushed into the stack. The next token is 'st' which means pop the top of the stack to the variable 't', form the expression 't=a[j]', and assign it to the data operation field of the current link (the first link). The address field of the link is also created: (blk_num,seq_num)=(2,0). The seq_num is then incremented to 1.

The next token is 'lj', and the resulting action is push 'j' into stack, followed by 1, which is also pushed into the stack. Next the operator '+' will cause the top two elements of the stack to be popped, and the expression 'j+1' is pushed back into the stack. The ';' indicates that the expression 'j+1' on top of the stack is an array index. The next token 'lj' causes 'j' to be pushed in the stack, the ':' indicates an array operation that pops the top element of the stack (in this case 'j'), uses it as an array index and assigns to it the next top element of the stack ('a[j+1]' in this example). The variable or array name (which is 'a') is also found as mentioned previously. The expression thus formed is 'a[j]=a[j+1]'. Note that ':' is of the pop operand type and thus the resulting expression is assigned to the statement in the current link, and the address formed is '(2,1)'. The seq_num is then incremented.

The algorithm continues in the same way to form the cell 'a[j+1]=t', and the delimiter ']' indicates the end of the block. Since the father of this block is not yet defined, it is left temporarily empty.

Now the next block, given below, is read:

```
[lj;a lj 1+;a <2ljd1 +sjs. ljli >1]s1
```

The blk_num is 1 and the seq_num is reset to 0. The statements 'lj;a lj 1 +;a' will result in forming the expressions 'a[j]' and 'a[j+1]' as the top two elements of the stack. The token '< 2' will form the condition a[j+1] < a[j] and assign it

```
a[1]=21
a[2]=19
a[3]=13
a[4]=12
a[5]=1
for (i=5; i>1; i--) {
   for (j=1; j<i); j++) {
      if (a[j]>a[j+1]) {
         t=a[j]
         a[j]=a[j+1]
         a[j+1]=t
      }
   }
}
```

(a)

```
21  1:a
19  2:a
13  3:a
12  4:a
 1  5:a
[lj;a st lj 1+;a lj:1ltlj 1+:a]s2
[lj;a lj 1+;a <2 ljd1+sj ljli>1]s1
[1dsj ljli>1 lid1-si li 1<0]s0
5dsi li 1<0
q
```

(b)

Figure 8 The C and BRPN descriptions of bubble sort algorithm

to the control operation field of the link. The address of this link is (current blk_num, current seq_num) which is (1,0). The true branch field gets (2,0) and the false branch field gets (1,1). The 'Father' of the called block (which is block 2) is (1,1). Since the father of block 2 is now defined, the generated code is updated accordingly. Finally, the generated CRTL code is:

```
(1,0) if a[j+1]<a[j] then (2,0)
(1,1) j<=j+1
(1,2) if i>j then (1,0)
(1,3) goto 'father'
(2,0) t<=a[j]
(2,1) a[j]<=a[j+1]
(2,2) a[j+1]<=t
(2,3) goto (1,1)
```

The complete bubble sort CRTL model generated from BRPN is given in Figure 9. Note that the stack operations outside the blocks are grouped into block number **–1**.

3.3 AHPL as a target RTL

The language AHPL[8] uses the convention that any digital system can be divided into a data part and a control part. The information about the data transfer and the control flow is stored in the form of a linked list. The corresponding RTL specification in AHPL can easily be obtained. In this section, a very brief overview of the AHPL language is presented, following which, the generation procedure of the AHPL code from CRTL is discussed.

```
      (-1,0)   a[1]=21.
      (-1,1)   a[2]=19.
      (-1,2)   a[3]=13.
      (-1,3)   a[4]=12.
      (-1,4)   a[5]=1.
 1.   (-1,5)   i<=5.
 2.   (-1,6)   if i>1 then (0,0).
 3.   (-1,7)   End.
 4.    (0,0)   j<=1.
 5.    (0,1)   if i>j then (1,0).
 6.    (0,2)   i<=i-1.
 7.    (0,3)   if i>1 then (0,0).
 8.    (0,4)   goto (-1,7).
 9.    (1,0)   if a[j+1]<a[j] then (2,0).
10.    (1,1)   j<=j+1.
11.    (1,2)   if i>j then (1,0).
12.    (1,3)   goto (0,2).
13.    (2,0)   t<=a[j].
14.    (2,1)   a[j]<=a[j+1].
15.    (2,2)   a[j+1]<=t.
16.    (2,3)   goto (1,1).
```

Figure 9 CRTL description of bubble sort algorithm

This section is not intended to detail the entire AHPL language but to mention only the basic constructs. We also restrict the discussion to the *SEQUENCE section* of the AHPL model. This is the section that models the finite state automaton.

The AHPL SEQUENCE section consists of a sequence of numbered steps, each step representing a state of the finite state machine. A step may have transfer statements '<=', connections to buses '=', or conditional/unconditional branches. Conditional branches are expressed as '$=>(f_1, f_2, \ldots, f_n)/(S_1, S_2, \ldots S_n)$', which read as, if condition f_i is true, then, in the next clock pulse, transfer control to Step S_i; else transfer to the following step in sequence. Unconditional branching to Step S_i is expressed as '$=>(S_i)$'. The convention in AHPL is that all transfers to registers and state transitions are assumed to take place at the trailing edge of the clock pulse, while transfers to buses are active during the entire duration of the clock. In addition, an abbreviated form for expressing combinational functions called CLUs (combinational logic units or functional units) is available. Complex combinational logic circuits can be modeled separately as CLUs, and then invoked in the sequential part of the description when needed. Examples of combinational logic units are comparator (COM), bus_function (BUSFN), binary decoder (DCD), incrementer (INC), and decrementer (DEC). The comparator is used to compare two vectors and has three outputs. In COM(I;J), I and J are binary vectors, the third output bit of the CLU, that is COM {2} is high if condition '$I < J$' is true (one of the other two outputs is high depending on whether $I > J$ or $I = J$). Reading from memory is done through the decode (DCD) and the bus_function (BUSFN) CLUs. The DCD unit decodes the given address and enables the corresponding memory location value. CLU BUSFN then routes the value of the enabled location to the destination. Thus a memory read is accomplished as follows,

```
destination_register<=BUSFN
      (MEMORY,DCD(address_register))
```

Writing to the memory is done in a similar way. The address is decoded using DCD and the corresponding memory location is enabled for writing. The instruction is

```
MEMORY*DCD(address_register)<=source.
```

For a more detailed overview of AHPL the reader may refer to Sait *et al.*[9]. The complete documentation on the language and its grammar are available in Masud[8].

With this introduction to AHPL, let us consider the memory conflict issue which is a by-product of compiling high-level language descriptions. Statements such as '$a[j] < a[j + 1]$' are possible in high-level descriptions. However, in AHPL it is not possible to read two locations of the memory bank simultaneously and compare their contents. Similarly, it is not possible to simultaneously read and write, i.e. move the contents of one memory location into another in a single step. To resolve this conflict, the action is divided into two steps and temporary registers are introduced as described in Figure 10. For the bubble sort example (Figure 9), there are two steps where we have memory conflicts: Steps 9 and 14. They are resolved by splitting Step 9 into 9.1 and 9.2 and

IF a memory variable has a conflict *then*

Split the statement into two or more statements ;
Use a new variable in one statement;
Refer to the new variable in the other one

END_IF

Figure 10 Algorithm for memory conflict resolution

```
MODULE   : SORT.
MEMORY   : A{8}<8>;I{3};J{3};T{8}.
MEMORY   : YY1{8}; YY2{8}.
EXINPUTS : RESET;CLOCK.
CLUNITS  : DCD{6};INC{3};BUSFN{12};COM3{3};COM8;DEC{3}.
BODY SEQUENCE:CLOCK.
  (-1,5)  1     I<=\1,0,1\.
  (-1,6)  2     =>COM3{2}(\0,0,1\;I)/(4).
  (-1,7)  3     DEADEND.
  ( 0,0)  4     J<=\0,0,1\.
  ( 0,1)  5     =>COM3{2}(J;I)/(9.1).
  ( 0,2)  6     I<=DEC(I).
  ( 0,3)  7     =>COM3{2}(\0,0,1\;I)/(4).
  ( 0,4)  8     => (3).
  ( 1,0)  9.1   YY1 <= BUSFN(A;DCD(J)).
  ( 1,0)  9.2   => COM8{2}(BUSFN(A;DCD(INC(J)));YY1)/(13).
  ( 1,1) 10     J<=INC(J).
  ( 1,2) 11     =>COM3{2}(J;I)/(9.1).
  ( 1,3) 12     => (6).
  ( 2,0) 13     T<=BUSFN(A;DCD(J)).
  ( 2,1) 14.1   YY2<=BUSFN(A;DCD(INC(J))).
  ( 2,1) 14.2   A*DCD(J)<=YY2.
  ( 2,2) 15     A*DCD(INC(J)) <= T.
  ( 2,3) 16     =>(10).
ENDSEQUENCE
CONTROLRESET(RESET)/(1).
END.
```

Figure 11 The bubble sort example after resolving memory conflicts

introducing temporary register YY1. While Step 14 is split into 14.1 and 14.2 and register YY2 is used. The generated AHPL model for the bubble sort is given in Figure 11.

4. OPTIMIZATION OF AHPL DESCRIPTIONS

Scheduling refers to the problem of assigning the operations to control steps in order to maximize concurrency, while satisfying existing resource constraints (hardware and time). Scheduling is an essential step of high-level synthesis. Besides deciding the correct ordering of the operations, it also determines the number of various hardware resources required by the allocation step. The outcome from the scheduling step depends entirely on two main features:

1. the scheduling objective, and
2. the existence of constraints, such as limit on resources.

In this work, we assume unlimited resources, and our objective is to obtain an RTL description with the smallest number of control steps (CSteps). Two types of optimization are considered: software optimization and hardware optimization. Other constraints such as limit on resources, and other objective functions such as reduction of cycle time can be accommodated.

4.1 Software optimization

Software optimization refers to compiler-like optimizations. The main techniques considered in this work consist of unconditional branch elimination and code factorization.

4.1.1 Unconditional Branch Elimination

Each transfer of control from a child block to its father results in an unconditional branch statement. These statements are extra and therefore are deleted. Their true branch addresses are copied into the true or the false branches of the calling statements. The algorithm used for unconditional branch elimination is shown in Figure 12 and has an $O(n^2)$ complexity, where n is the number of statements in the AHPL code.

The algorithm initially scans every statement looking for unconditional branches. A second scan is done by the algorithm for branch adjustment. For example, consider the following code segment:

```
10  J<=INC(J); => (5).
:
15  A*DCD(INC(J))<=T.
16  => (10).
```

Statement 16 is an unconditional branch. It is deleted and Statement 15 is adjusted as follows:

```
15 A*DCD(INC(J))<=T; => (10).
```

4.1.2 Code Factorization

The canonical RTL description may contain steps with similar expressions. These redundant expressions are easy to

For each statement in the code *do*
If a statement *i* is a dead statement *then*

Scan the code for any statement *j* branching to *i* then:
Modify *j* to branch to the address that *i* is branching to

End_If
End_For

Figure 12 Unconditional branch elimination

For each statement *i* in the code *do*

Scan the code for any statement *j* that is similar to *i* then do:
delete *j* and modify the necessary branches

End_For

Figure 13 Code factorization

identify and factor out. The algorithm to do this is given in Figure 13 and has an $O(n^2)$ complexity where n is the number of statements in the AHPL code. Two scans are carried out by the algorithm. One to pick the statement and the other to check for identical statements and delete them. The branches of the calling statements are adjusted accordingly. For example consider the following code segment:

```
2   =>COM 2 (0,0,1;I)/(4).
:
6   I<=DEC(I).
7   =>COM 2 (0,0,1;I)/(4).
```

Statements 2 and 7 have similar expressions. In this case, Statement 7 is deleted, and the branch out of Statement 6 is adjusted to point to Statement 2 as follows:

```
6   I<=DEC(I); =>(2).
```

Example

To illustrate the optimization of this phase, consider the AHPL code of the bubble sort given in Figure 9. The resulting code after software optimization is given in Figure 14. The order of applying these optimization techniques may give different results. However it is difficult to say which order will lead to superior optimization, as this depends on the problem instance. In this work, unconditional branch elimination is applied first*. Steps 8, 12 and 16 are unconditional branches. Thus they are deleted. The branchings in Steps 7, 11 and 15 are modified to include 3, 6 and 10 respectively. In code factorization phase, Steps 2 and 7 are found similar. Hence Step 7 is deleted and Step 6 is modified to branch to Step 2. Moreover, as Step 3 is the successor of Step 2, it is not necessary to include 3 in the set of output branch addresses of Step 2. Similarly, Steps 5 and 11 are redundant and Step 11 is deleted. Modification in the code is done in a similar manner. The control steps in the resulting code have been reduced by five control steps. The optimized AHPL description is given in Figure 14.

4.2 Hardware specific optimization

In AHPL, register transfers take place at the trailing edge of the clock. That is, changes to registers within a control step are not effective at the control step, but only one step later. This important property† of the language permits further

*Experiments suggest that elimination of unconditional branch first results in better optimization.

†This property is shared with most RTL languages in one form or another.

```
MODULE   : SORT.
MEMORY   : A{8}<8>;I{3};J{3};T{8}.
MEMORY   : YY1{8}; YY2{8}.
EXINPUTS : RESET;CLOCK.
CLUNITS  : DCD{6};INC{3};BUSFN{12};COM3{3};COM8;DEC{3}.
BODY SEQUENCE:CLOCK.
  (-1,5)  1     I<=\1,0,1\.
  (-1,6)  2     =>COM3{2}(\0,0,1\;I)/(4).
  (-1,7)  3     DEADEND.
  ( 0,0)  4     J<=\0,0,1\.
  ( 0,1)  5     =>COM3{2}(J;I)/(9.1).
  ( 0,2)  6     I<=DEC(I); => (2).
  ( 1,0)  9.1   YY1<=BUSFN(A;DCD(J)).
  ( 1,0)  9.2   =>COM8{2}(BUSFN(A;DCD(INC(J)));YY1)/(13).
  ( 1,1) 10     J<=INC(J); =>(5).
  ( 2,0) 13     T<=BUSFN(A;DCD(J)).
  ( 2,1) 14.1   YY2<=BUSFN(A;DCD(INC(J))).
  ( 2,1) 14.2   A*DCD(J)<=YY2.
  ( 2,2) 15     A*DCD(INC(J))<= T; => (10).
ENDSEQUENCE
CONTROLRESET(RESET)/(1).
END.
```

Figure 14 AHPL description of bubble sort algorithm after applying software optimization

optimization to the target RTL description. We refer to optimization techniques that exploit this AHPL feature as hardware-specific optimization techniques.

Since we assume unlimited resources, minimizing the number of CSteps is equivalent to maximizing parallelism, i.e. assigning as many statements as possible to the same CStep. This will achieve other objectives as well: (1) it eliminates multiplexers, (2) it results in a smaller controller, and (3) it provides more opportunity for logic minimization. In Figure 15a, an AHPL model of a digital system is given. The control part of this AHPL model is shown in Figure 15b. Merging different CSteps (see Figure 15c) produces a smaller control circuit, as shown in Figure 15d.

Hence, as far as the logic in the data path is concerned, the maximization of concurrency increases the chance of eliminating redundant logic. Silicon compilers like the AHPL silicon compiler run logic optimization on their input models. The optimization is applied only to the logic within a CStep rather than the logic in the whole model. By clustering as many statements as possible in a single CStep, we increase the chance for logic optimization. Thus, leading to the minimization of overall silicon area. This has been confirmed by experimental results as will be discussed in Section 6.

Hardware specific optimization algorithms are developed to accomplish the above task. Central to these algorithms is a template which is the subject of the following section.

4.2.1 Template

Recalling the general format of CRTL statements (see Section 3), each statement in the CRTL code is mapped into the following template and tagged with its reference number:

$$< Bin_i, O_i, I_i, Bout_i >$$

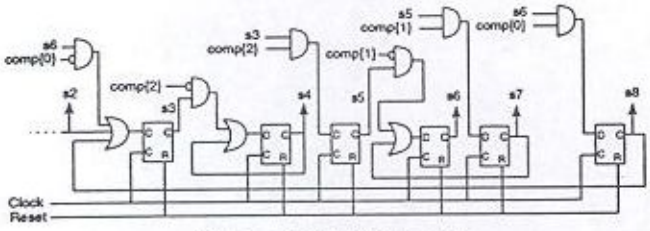
where

Bin_i represents the set of all predecessor statement numbers of statement i,

O_i the set of all output variables in statement i,

```
3   => (comp{2}(x,y),-comp{2}(x,y)) / (5,4).
4   => (4).
5   => (comp{1}(x,y),-comp{1}(x,y)) / (7,6).
6   => (comp{0}(x,y),-comp{0}(x,y)) / (8,3).
7   y<= INC(ADD(y,-x));  =>  (6).
8   x<= INC(ADD(x,-y));  =>  (3).
```

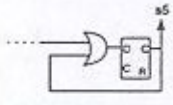
(a) AHPL model



(b) Control unit of above model

```
5   y*comp{1}(x,y)<= INC(ADD(y,-x));
    x*comp{0}(x,y)<= INC(ADD(x,-y));    =>  (5).
```

(c) AHPL model after merging CSteps (Optimized model)



(d) Control unit of optimized model

Figure 15 Example of merging different Csteps

I_i the set of all input (used) variables in statement i, and $Bout_i$ the set of all successor statement numbers ($|Bout_i| > 0$). If $|Bout_i| = 0$ then statement i is a dead end.

The sets O_i and I_i for the ith statement are formed from the variables found either in the data or the condition operations. Two optimization techniques are proposed: *loop transformation* and *switch transformation*.

4.2.2 Loop Transformation

Loops can be *predicate loops* or *counter loops*. For a predicate loop, the execution is controlled by a boolean expression. Changes in the logic value of this expression are controlled by the statement in the body of the loop. The optimization involved in these loops is similar to the switch transformation which is discussed in the next section.

A counter loop, in general, consists of an initialization step, a compare step, a body, and an adjust step. The number of executions in a loop is controlled by a counter variable (cv). Every time an iteration is completed, cv is adjusted accordingly. In AHPL, the adjust and compare steps can be performed in the same CStep. The reason is that in AHPL, register transfers occur at the trailing edge of the clock. This means that the new value of a variable is set at the trailing edge. During the clock, the old value is unchanged. We refer to the operation of merging the adjust and compare steps as *loop transformation*. Figure 16 illustrates this merging. However, one has to be careful because this merging

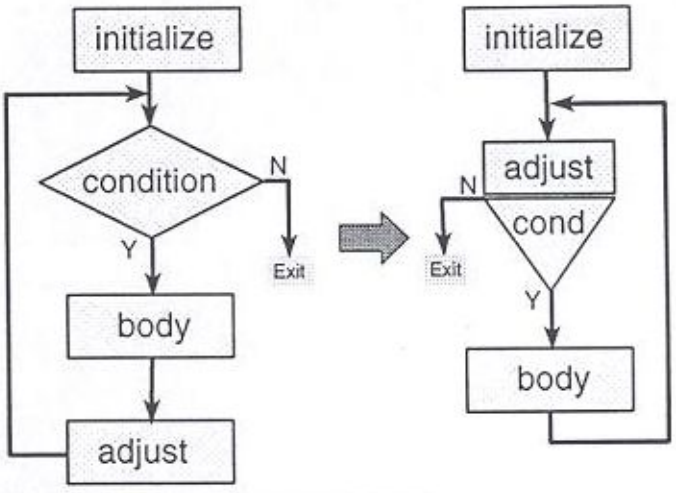


Figure 16 The definition of loop transformation

requires that cv be adjusted inside as well as outside the loop. The adjustment is essential as cv holds the next value at the beginning of the loop rather than at the end.

The variable cv appears in both the data and the control expressions after this merging. Because of the aforementioned property of AHPL register transfers, cv in the control part of the compare step stays unchanged. The new value of cv is set in the next clock cycle.

To adjust the value of cv inside and outside the loop, we need to backtrack to all locations where cv is invoked. The expressions that use cv are replaced with new expressions containing the adjusted value of cv. That is, the effect of the adjustment step is undone. For example, assume that the adjust step for a loop contains '$cv = cv + x$'. Then cv in any expression inside or outside the loop is replaced with '$cv - x$'. We shall call this adjustment step as $ADJUST(cv)$.

The loop transformation algorithm is given in Figure 17. The algorithm is executed when the adjust step is located. Let S_i be a particular statement and I_i and O_i be respectively, the set of input variables and the set of output variables of statement S_i.

Definition 1

A statement S_i is a loop adjustment step if and only if $I_i \cap O_i \neq \emptyset$ and $I_i \cap O_i = \{cv\}$ where cv is the loop controlling variable.

Once an adjust step is identified, a search for the compare step is initiated. Then the compare step is located, and a backtrack procedure is invoked to find the places where cv is used. The variable cv is then adjusted as described above.

To illustrate the algorithm, consider the AHPL model for the bubble sort example of Figure 14. The flowchart representing the partially optimized AHPL state machine is shown in Figure 19.

Loop transformation is illustrated in Figure 20. The variable 'j' is cv. The shaded region in this figure represents the body of the loop, and is executed if the condition '$i > j$' is true (Step 5). If the condition is false then control branches to Step 6. The adjustment of cv is done in Step 10 '$j <= j + 1$'. Figure 20b illustrates the merging of Steps 5 and 10, and the adjustment of the value of j both inside the loop and at Step 6 where j is decremented.

```
IF (O_i ∩ I_i ≠ Ø & |O_i ∩ I_i| = 1) THEN
   cv = O_i ∩ I_i          \* cv is the controlling variable *\
   Search for S_j | cv ∈ Cond(S_j)
   IF S_j is located THEN
      BackTrack(S_i, ADJUST(cv))
      Merge S_i into S_j
   END_IF
END_IF
```

Figure 17 Algorithm for loop transformation

```
Procedure BackTrack(S_i , ADJUST(cv))
  Begin_Procedure
     FOR each S_k ∈ Bin(S_i)
        BackTrack(S_k , ADJUST(cv))
        IF cv ∈ O_i ∪ I_i THEN
           Replace cv with ADJUST(cv)
        END_IF
     END_FOR
  End_Procedure
```

Figure 18 Back track procedure

Lemma 1

A loop transformation reduces the number of Control Steps by at least one CStep.

Proof

A loop has a compare step and an adjust step. Merging the adjust step with the compare step results in reducing the total number of CSteps by one. The adjustments that may be required to statements where cv is used outside the loop do not cause any increase in the number of CSteps. ❐

4.2.3 Switch Transformation

A statement is a *switch* if it branches to more than one statement. A condition is associated with each branch. The branch is taken (executed) if the corresponding condition is true.

If a statement is switching to a number of statements, then these statements can be merged into one statement. The conditions can either be inclusive or mutually exclusive. In mutually exclusive conditions, one and only one condition is true. Whereas, in inclusive conditions, none or at least one condition is true.

Mutually exclusive conditions: In the case of mutually exclusive conditions, the switch forks to at most 2^n branches, where n is the number of Boolean variables forming the conditions. To illustrate this transformation on AHPL descriptions, consider the following general AHPL code:

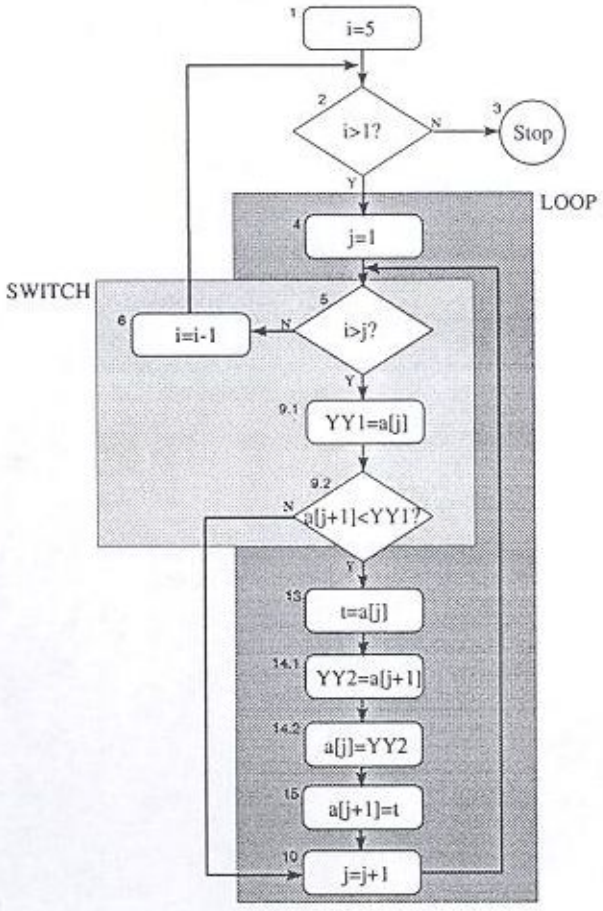


Figure 19 Partially optimized AHPL model for bubble sort example

Step#	Data transfers	Control transfers
$k.$	$stmt_k$;	$\Rightarrow (m_1, m_2, \ldots, m_{2^n})/(s_1, s_2, \ldots, s_{2^n})$.
$\vdots$	$\vdots$	$\vdots$
$s_1.$	$stmt_{s_1}$;	$\Rightarrow (x_1, x_2, \ldots, x_l)/(t_1, t_2, \ldots, t_l)$.
$s_2.$	$stmt_{s_2}$;	$\Rightarrow (y_1, y_2, \ldots, y_p)/(u_1, u_2, \ldots, u_p)$.
$\vdots$	$\vdots$	$\vdots$
$s_{2^n}.$	$stmt_{s_2^n}$;	$\Rightarrow (z_1, z_2, \ldots, z_q)/(v_1, v_2, \ldots, v_q)$.

In AHPL, more than one register transfer operation can take place in a single control step. Moreover, these register transfers can be conditionally controlled. A particular data transfer operation will only be executed if the corresponding condition is true. Therefore, the above code can be transformed to a more compact form as follows:

$$k. \quad stmt_k; stmt_{s_1} * m_1; stmt_{s_2} * m_2; \ldots; stmt_{s_2^n} * m_{2^n};$$
$$\Rightarrow (x_1 m_1, \ldots, x_l m_1, y_1 m_2, \ldots, y_p m_2, \ldots, z_1 m_{2^n}, \ldots, z_q m_{2^n})/(t_1, \ldots, t_l, u_1, \ldots, u_p, \ldots, v_1, \ldots, v_q).$$

The conditional branch in the transformed statement is the logical *and* operation between the switch conditions and the conditions of the branches. Figure 21 shows the switch trans-

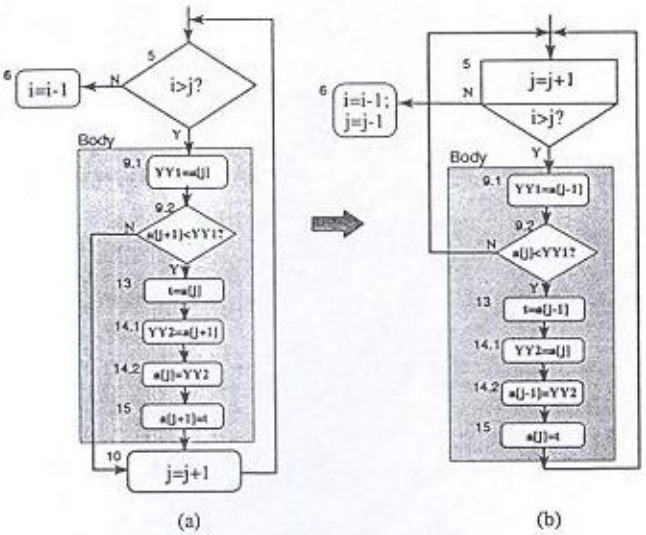


Figure 20 Flow chart illustrating loop transformation

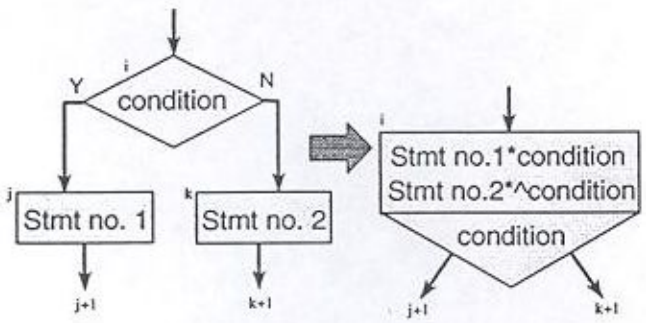


Figure 21 Fork transformation of an it If-Then-Else statement

formation technique applied to the *If-Then-Else* statement.

Lemma 2

A memory conflict free switch with n-boolean variables forming 2^n mutually exclusive conditions is reduced by 2^n CSteps.

Proof

The total number of CSteps forming this switch is $2^n + 1$ (the condition statement + 2^n branches). If there is no memory conflict between the branches and the switch, then these branches are merged into the switch. This results in a code reduction of 2^n Csteps. ❐

Lemma 3

A switch with n-boolean variables forming 2^n mutually exclusive conditions is reduced by at least $2^n - 1$ Csteps.

Proof

As in the above lemma, 2^n branches are formed. In the case of memory conflict free condition we achieve a reduction by 2^n CSteps. In the presence of memory conflicts, two CSteps are needed, one to evaluate the conditions and the other to perform the corresponding register transfer operation. A total of $2^n - 1$ CSteps are saved. ❐

Inclusive conditions: Unlike mutual exclusive conditions, inclusive conditions are independent. That is, the variables forming the conditions are not related. Moreover, the number of conditions forming the conditional branch in an AHPL statement is not limited. Thus, if none of the conditions is true, the next AHPL statement in sequence is executed. To show the optimization involved in this case, consider the AHPL statements below:

Step#	Data transfers	Control transfers
$k.$	$stmt_k$;	$\Rightarrow (m_1, m_2, \ldots, m_i)/(s_1, s_2, \ldots, s_i).$
$k+1.$	$stmt_{k+1}$;	$\Rightarrow (\ldots).$
$\vdots$	$\vdots$	$\vdots$
$s_1.$	$stmt_{s_1}$;	$\Rightarrow (x_1, x_2, \ldots, x_l)/(t_1, t_2, \ldots, t_l).$
$s_2.$	$stmt_{s_2}$;	$\Rightarrow (y_1, y_2, \ldots, y_p)/(u_1, u_2, \ldots, u_p).$
$\vdots$	$\vdots$	$\vdots$
$s_i.$	$stmt_{s_i}$;	$\Rightarrow (z_1, z_2, \ldots, z_q)/(v_1, v_2, \ldots, v_q).$

Because $m_1, m_2, \ldots, m_i$ are independent conditions, there is a probability that none of them is true. Thus, a correct transformation of the above $i + 2$ CSteps is as follows:

$$k. \quad stmt_k; stmt_{s_1} * m_1; stmt_{s_2} * m_2; \ldots; stmt_{s_i} * m_i;$$
$$\Rightarrow (x_1 m_1, \ldots, x_l m_1, y_1 m_2, \ldots, y_p m_2, z_1 m_i, \ldots, z_q m_i, \overline{(m_1 + \ldots + m_i)})/(t_1, \ldots, t_l, u_1, \ldots, u_p, \ldots, v_1, \ldots, v_q).$$

Now, statement $k + 1$ is part of the control transfer code. If none of the conditions is true, control branches to step $k + 1$.

Lemma 4

A memory conflict free switch with k inclusive conditions is reduced by k + 1 CSteps.

Proof

For a switch with k independent conditions, k different statements can be branched to. If none of the conditions is true then a branch to the following statement takes place. A total of $k + 1$ branches are available. If none of the branches has memory conflict, then $k + 1$ branches are merged into the switch. This reduces the code by $k + 1$ CSteps. ❐

Lemma 5

A switch with k inclusive conditions is reduced by at least 1 CStep.

Proof

Since the conditions are independent, one or more branches can be taken at the same time. Moreover, all the branches can be taken when all conditions are true. In the extreme case of memory conflicts between all conditions, none of the branches are mergeable. The reason is that the conditions may all be true. If none of the conditions is true, the following statement is executed. As the condition of this statement is disjoint with the conditions of the branches, this statement is merged with one of the branches. Therefore, at least one CStep is saved.

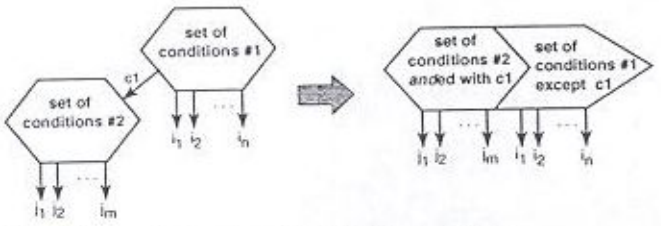


Figure 22 Cascade transformations at two level switches

The switch transformation, in the case of inclusive conditions is more general than the case of mutual exclusive conditions. In fact, mutual exclusiveness is a special case of inclusive conditions.

The switch transformation is applied recursively. Therefore, a branch of a switch statement can also be to another switch statement. For example, in the case of two-level switching, the branches in the second level are merged into the switch of the first level. A new condition set is formed from the two sets. The first set is the set of conditions in the first level switch excluding the condition that causes branching to the second level switch. The other set is formed by *anding* the excluded condition with the set of conditions in that branch. This is illustrated in Figure 22.

The switch transformation algorithm is given in Figure 23. The algorithm is called by the main program once a statement with at least a condition expression is found. The set of conditions (C) in this statement is formed. For every condition $c_k \in C$, if the corresponding statement has no memory conflict, then the statement can be merged into the switch statement. The execution of the child statement is controlled by the true value of c_k. The algorithm continues scanning AHPL statements and merging them whenever possible.

To illustrate how the algorithm works, consider the flow chart of the bubble sort example after loop transformation (Figure 24a). Two switches are located in the flow chart of Figure 24a. One in Step 2 and the other in Step 5. From Step 2, C contains $i > j$, $\sim (i > j)$. If the condition '$i > j$' is true, we proceed with Step 4. Step 3 is called when the condition '$\sim (i > j)$' is true. Step 4 has no memory conflict with Step 2. The procedure is recursively called until all conditional expressions in Step 4 are processed. Thus, Step 4 is merged into Step 2, on the condition that '$i > j$'. Then Step 3 is merged

```
Procedure Switch_Transformation(S_i)
  Begin_Procedure
    Form the set of conditions C
          C={c1,c2,...,cj}; j≥ |Bout_i|
    FOR   each c_k ∈ C
      IF (S_k has no memory conflict with S_i) THEN
        Switch_Transformation(S_k)
        Merge S_k * c_k into S_i
      END_IF
    END_FOR
  End_Procedure
```

Figure 23 Algorithm for switch transformation

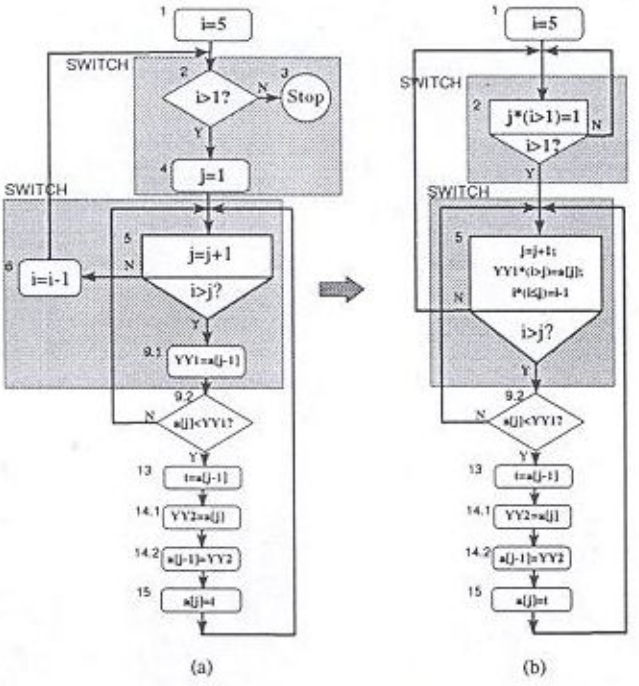


Figure 24 Flow chart illustrating switch transformation in bubble sort

into Step 2. A dead end (stop) is achieved by a transfer back to the same state.

Applying the algorithm to the switch at Step 5, a conditional branch forks to Steps 6 and 9.1. Data transfers in these Steps are converted to destination controlled conditional transfers in Step 5 of Figure 24b. Therefore, Step 6 which contains the transfer '$i \Leftarrow i-1$' is moved to Step 5 and is executed if '$i \leq j$'. The same transformation is applied to the transfer in Step 9.1. Figure 24b shows the flow chart of the bubble sort AHPL model after switch transformation

5. A COMPREHENSIVE EXAMPLE

To illustrate the entire synthesis process, we use the greatest common divisor (GCD) benchmark test[10]. The algorithm for GCD is

```
x=4
y=5
while(x! = y) {
      if(x < y) {y = y − x}
      if(x > y) {x = x − y}
}
```

The translation of the above *GCD* behavioral description results in the following BRPN intermediate form

```
4sx
5sy
[lylx − sy]s1
[lylx − sx]s2
[lylx > 1lxly < 2lxly! = 0]s0
lxly! = 0
q.
```

The second step is to obtain the CRTL code from the BRPN description. The outcome of this step is the following:

```
(-1,0) 1: x=4
(-1,1) 2: y=5
(-1,2) 3: if y!=x then 5
(-1,3) 4: Deadend
( 0,0) 5: if y>x then 9
( 0,1) 6: if y<x then 11
( 0,2) 7: if y!=x then 5
( 0,3) 8: goto 4
( 1,0) 9: y=y-x
( 1,1) 10: goto 6
( 1,2) 11: x=x-y
( 1,3) 12: goto 7
```

Then, following the application of software optimization, Steps 8, 10 and 12 which are unconditional branches are deleted. Steps 7, 9 and 11 are modified to branch to Steps 4, 6 and 7 respectively. Steps 3 and 7 are similar. Thus, Step 7 is deleted and the code is modified. The partially optimized code is as follows:

```
(-1,0) 1: x=4
(-1,1) 2: y=5
(-1,2) 3: if y!=x then 5
(-1,3) 4: Deadend
( 0,0) 5: if y>x then 9
( 0,1) 6: if y<x then 11 else 3
( 1,0) 9: y=y-x; goto 6
( 1,2) 11: x=x-y; goto 3
```

The repetitive application of the switch transformation leads to the merging of Steps 3, 5 and 6 into one CStep. Thereafter, the register transfer operation in Step 9 is translated into a destination controlled conditional transfer '$y * (y > x) = y - x$'. Similarly, Step 11 is transformed into '$x * (y < x) = x - y$'. Since there is no memory conflict in Steps 1 and 2, these are merged. Figure 25 shows these transformations. The GCD algorithm finishes when y and x are equal. Therefore, a *finish* signal is introduced. This signal is true only when $y = x$.

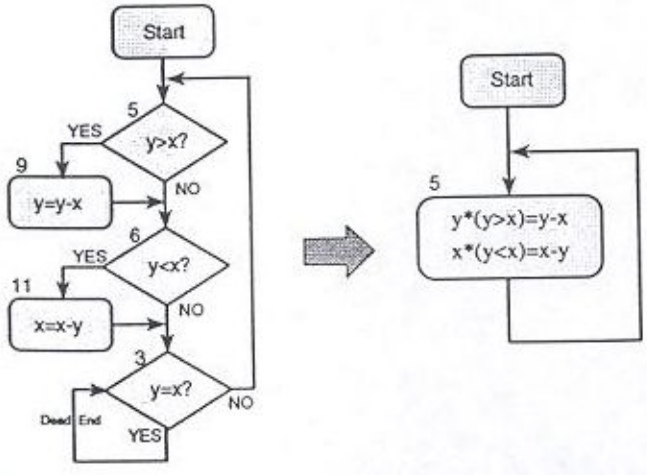


Figure 25 Switch transformation in the GCD example

The optimized AHPL model for the *GCD* example is listed below. All conditional transfers are destination controlled transfers.

```
1  x = 4; y = 5; => (5).
5  y * (x > y) = y - x; x * (x < y) = x - y;
   finish * (y = x) = \1\; => (5).
```

6. EXPERIMENTAL RESULTS AND DISCUSSION

The high-level synthesis system described in this work proceeds in three steps: BRPN translation, CRTL generation, and CRTL optimization. These tasks are integrated as illustrated in Figure 1.

The system has been implemented in the C programming language. The final step in this system is the generation of the AHPL code from the optimized CRTL. The AHPL code is then fed to an AHPL silicon compiler to generate the physical description (layout)[9]. The VLSI layout of the control unit of the bubble sort circuit is given in Figure 26.

The results of generating concise RTL code of digital systems modeled in high-level language are presented here. Benchmarks from Dutt and Ramachandran[10] are used. The benchmark circuits used are traffic light controller (TLC), greatest common divisor (GCD), and differential equation (DiffEq). The bubble sort (BubSrt) model is also used to study the situation of having memory conflicts in the AHPL code. The generated circuits are compared with those reported elsewhere[11-13]. The comparison is with respect to the number of CSteps required.

During optimization, transformation algorithms search for possible merging of control steps among the unoptimized RTL statements. This optimization process reduces the number of control steps (*flip-flops*) as well as the number of logic components used in the control logic; the overall logic may also be minimized. In Table 1 the first column (*Before optimization*) shows the hardware resources required by the control logic in the unoptimized RTL description. The columns

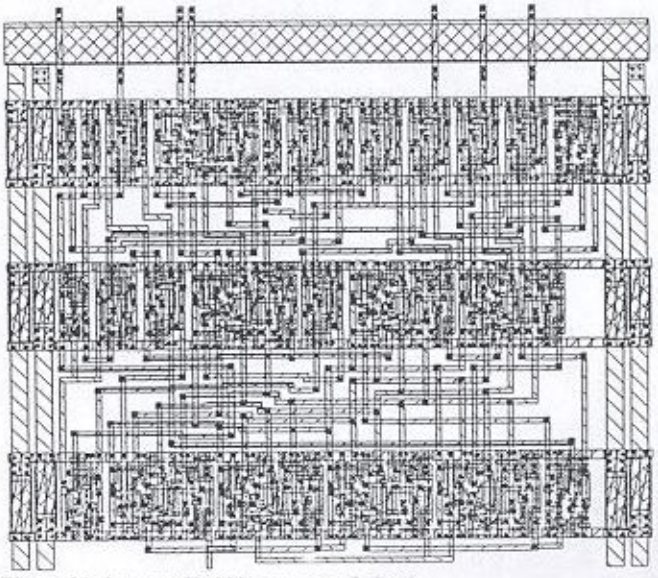


Figure 26 Layout of bubble sort control circuit

Table 1 Control logic in uncompacted and compacted RTL

Circuit	*Before optimization*		*After optimization*	
	CSteps	Logic Units	CSteps	Logic Units
TLC	63	36-And,17-Or,3-Comp	2	1-Or
GCD	9	8-And,4-Or,4-Comp	2	1-Or
DiffEq	19	4-And,2-Or,2-Comp	2	1-Or
BubSrt	18	10-And,4-Or,5-Comp	6	2-And,3-Or,3-Comp

Table 2 The number of CSteps at different stages of the high-level synthesis system

Circuit	*Unoptimized*	*Software*	*Hardware*
TLC	63	47	2
GCD	9	7	2
DiffEq	19	17	2
BubSrt	18	14	6

titled (*CSteps*) hold the number of control steps which correspond to the number of *flip-flops*. The required logic is given in the columns titled *Logic Units*. The logic components used are: 4-bit comparator (Comp), and some logic gates (And, Or). The second column (*After optimization*) shows the resources allocated based on the optimized code.

Software as well as hardware optimization techniques are performed on each circuit. Exploiting hardware special features has resulted in more optimized models. This is illustrated in Table 2.

In the *TLC* benchmark test, the unoptimized code has 63 CSteps. The application of software optimization reduced the number of CSteps in the code by 16. While the application of hardware specific optimization resulted in a 2-CStep AHPL model.

The results obtained in this work are compared with those reported in literature. In Table 3, the columns titled *LBS* (loop based scheduling), *PBS* (path based scheduling), and *DLS* (dynamic loop scheduling) give the number of states obtained by the scheduling algorithms reported in Al-Sukhni[11], Camposano[12] and O'Brien *et al.*[13], respectively. The last column titled HSO (hardware specific optimization) in the table, holds the number of states obtained by the application of the optimization techniques developed in this work. The experimental results show a reduction in the number of states ranging from 50% for DiffEq to 75% for TLC. This reduction was mainly a consequence of hardware specific optimization.

Another measure to test the quality of the optimization techniques is to calculate how much reduction is achieved in the total area of the chip. Table 4 illustrates this.

Table 4 The areas of test circuits at different stages of the high-level synthesis system

Circuit	*Unoptimized*	*Software*	*Hardware*	*-%*
TLC	653,632	562,832	223,644	65.8
GCD	971,152	960,480	598,096	38.4
DiffEq	2,712,544	2,602,112	2,486,112	8.3
BubSrt	1,813,312	1,586,848	1,427,728	21.3

The numbers in the first three columns are in λ^2. The figures in the first column are the areas of the test circuits before any optimization is performed. The second column titled *Software* holds the areas after the application of software optimization. The column *Hardware* shows the areas of the optimized circuits after both software and hardware optimizations. The relative reduction in each model is shown in the last column (–%). For example, the total area of the optimized AHPL code for the TLC benchmark test is 223,644 λ^2. The area of the unoptimized model is 653,632 λ^2. This gives a percentage decrease in the total area by 65.8%. These numbers were obtained from the AHPL silicon compiler[9].

In Section 4.2, we stated that the chance for the silicon compiler to perform logic optimization increases as more logic is combined in a single CStep. Table 5 gives the percentage decrease in area at different stages of our high-level synthesis system.

The figures in each column present the percentage decrease in area when the logic optimization of AHPL silicon compiler is performed. For example, in the unoptimized TLC model, the AHPL silicon compiler accomplished 57.2% reduction in the total area. This reduction is due to logic optimization only. The application of software optimization has reduced the number of CSteps as discussed earlier. The reason is that, when the logic per CStep is increased, the chance for the AHPL silicon compiler to optimize hardware and reduce the circuit area improves.

Finally, the application of hardware specific optimization has further reduced the number of CSteps. Thus more logic is combined into a single CStep. A reduction of 74.8% in the circuit area was obtained.

Table 3 Experimental vs literature results

Circuit	*PBS*	*DLS*	*LBS*	*HSO*
TLC	8	7	5	2
GCD	2	2	2	2
DiffEq	4	-	3	2
BubSrt	-	-	-	6

Table 5 The logic optimization performed by AHPL silicon compiler

Circuit	*Unoptimized*	*Software*	*Hardware*
TLC	57.2%	57.7%	74.8%
GCD	13.6%	13.8%	16.8%
DiffEq	20.2%	20.1%	20.6%
BubSrt	13.1%	19.8%	22.8%

The current implementation assumes unlimited resources and a single port memory. With reasonable effort, the system can be upgraded to accept user specified constraints with respect to clock cycle, hardware resources, chaining, etc. Such user specified constraints can be taken into consideration during the CRTL optimization. For example, the decision of chaining CSteps is dictated by the constraint on the clock frequency. Also, when merging CSteps, one has to check the availability of required hardware resources.

All heuristics used are of polynomial complexity (quadratic at most), which allows to quickly generate an optimized AHPL model of a given behavioral input. Run times for these examples were in the order of seconds. Experimental results seem to indicate that the number of CSteps in the synthesized models could constitute an accurate estimation of the minimum number of steps. More extensive experimentation is required to validate this observation.

7. CONCLUSION

In this paper we presented a high-level synthesis system which accepts as input a behavioral specification in a subset of the C language and generates optimized RTL descriptions in the AHPL language. We introduced a new stack intermediate form expressed in blocked reverse polish notation (BRPN). BRPN serves as a primary intermediate form from which other internal representations are extracted. The simplicity of the stack abstract data type made the generation to/from BRPN very efficient and easy. The target AHPL description is derived in two steps. First a canonical RTL description, where each step is limited to one statement, is extracted from the BRPN. Then, in the second step, the CRTL is optimized to produce a compact AHPL description with minimum number of CSteps. Besides the well known compiler-like optimization techniques, our scheduler exploits the hardware specific features of the AHPL language to perform suitable optimizations, yielding a schedule with minimum number of CSteps. Experiments on benchmark tests show sizable reduction in the number of CSteps compared to other reported systems. In current implementation, we assume unlimited hardware resources and a single port memory. Upgrades to allow the specification of resource constraints can easily be accommodated; instead of checking for memory conflicts only, one also has to check for other resource conflicts.

ACKNOWLEDGMENTS

Authors acknowledge support of King Fahd University of Petroleum and Minerals under KFUPM Project # COE/SYNTHESIS/145.

REFERENCES

1. **Tanka, T, Kobayashi, T and Karatsu, O** 'HARP: Fortran to Silicon', *IEEE Trans. CAD*, Vol 8 No 6 (June 1989) pp. 649–660.
2. **Trickey, H** 'Flamel: A High-Level Hardware Compiler' *IEEE Trans. CAD*, Vol 6 No 2 (March 1987) pp 259–269.
3. **Camposano, R and Rosenstiel, W** 'Synthesizing Circuits From Behavioral Descriptions', *IEEE Trans. CAD*, Vol 8 No 2 (February 1989) pp. 171–180.
4. **McFarland, M C, Parker, A C and Camposano, P** 'The High-Level Synthesis of Digital Systems', *IEEE Proc.*, Vol 78 No 2 (February 1990) pp. 301–318.
5. **Parker, A C** 'Automated Synthesis of Digital Systems', *IEEE Design and Test* (November 1984) pp. 75—81.
6. **Tseng, C-J and Siewoirk, D P** 'Automated Synthesis of Data path in Digital Systems', *IEEE Trans. CAD*, Vol 5 No 3 (July 1986) pp. 379–395.
7. **Pangrle, B-M and Gajski, D D** 'Design Tools for Intelligent Silicon Compilation', *IEEE Trans. CAD*, Vol 6 No 6 (November 1987) 1098–1112.
8. **Masud, M** Modular Implementation of a Digital Hardware Automation System, PhD dissertation, Department of EE, University of Arizona (1981).
9. **Sait, S M, Benten, M S and Khan, A M** 'Designing ASICs with UAHPL', *IEEE Circuits and Devices Magazine* (March 1995) pp. 14–24.
10. **Dutt, N and Ramachandran, C** Benchmarks for the 1992 High Level Synthesis Workshop, Technical Report no. 92-107 (October 1992).
11. **Al-Sukhni, H F** C-based High-Level Synthesis System, Master Thesis, King Fahd University of Petroleum and Minerals (January 1994).
12. **Camposano, R** 'Path-Based Scheduling for Synthesis', *IEEE Trans. CAD*, Vol 10 No 1 (January 1991) pp. 85–93.
13. **O'Brien, K, Rahmouni, M and Jerraya, A A** A VHDL-Based Scheduling Algorithm For Control-Flow Dominated Circuits IMAG/TIM3 Technical Report (1992).